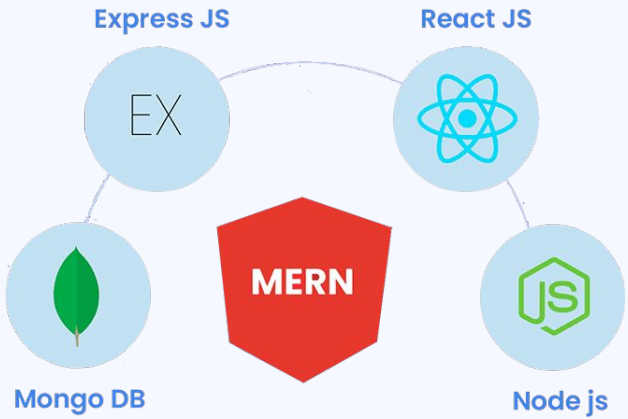


Full Stack Development Using MERN

Day 02 - 28th April 2024

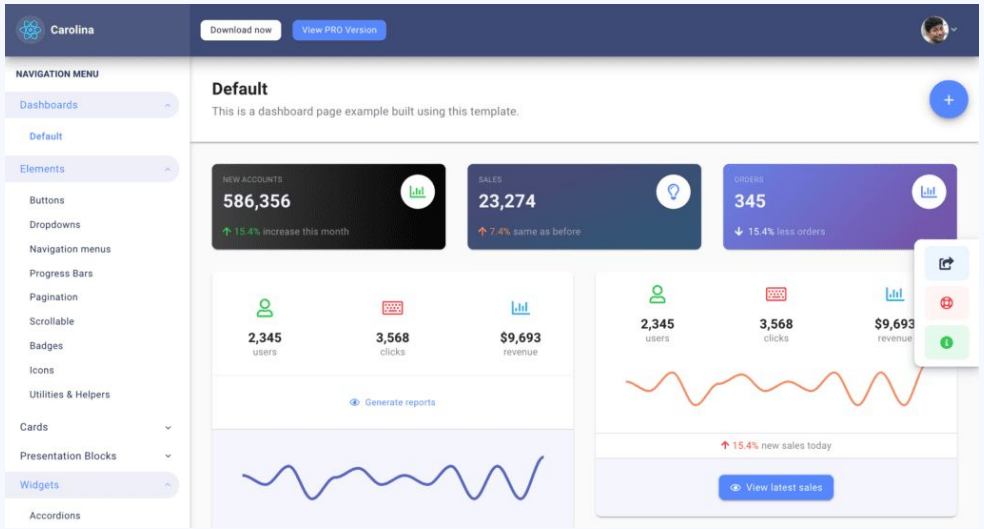
Lecturer Details:

Chanaka Wickramasinghe
cmw@ucsc.cmb.ac.lk
0771570227

Anushka Vithanage
dad@ucsc.cmb.ac.lk
071 527 9016

Prameeth Maduwantha
bap@ucsc.cmb.ac.lk
077 288 8098

Integrating External Libraries and UI Frameworks



Styled Components in React.js

- Styled-components is a library that allows you to write CSS in JS while building custom components in React.js.

- There are multiple options you can go with to style a React application. But the CSS in JS technique is good approach, where you write the CSS code right in the JavaScript file.

- Styled-components takes this approach.

Cascading Style Sheets - CSS

- CSS stands for "Cascading Style Sheets."
- It is a stylesheet language used to describe the visual presentation of a document written in HTML or XML.
- CSS allows web developers to control the appearance of elements on a webpage, including colors, fonts, spacing, positioning, and more.

CSS Syntax

- Selector: This specifies which HTML elements the rule applies to. For example, h1 selects all <h1> headings, and .class selects all elements with a specific class.
- Property: This defines the aspect of the element you want to style, like color, font-size, or margin.
- Value: This assigns a value to the property, such as red for the color or 20px for the font size.

```
selector {
  property: value;
  /* Additional properties and values */
}
```

Types of Selectors

- Element Selector: Targets specific HTML elements. **(<p>, <h1>)**
- Class Selector: Selects elements with a specific class attribute. **(button, p.home)**
- ID Selector: Selects a unique element with a specific ID attribute, **(#header, #para)**
- Descendant Selector: Selects an element that is a descendant of another element **(ul li selects all <li> elements within a <ul>.)**
- Pseudo-class Selector: Selects elements based on their state, **(hover for styling links when hovered over)**

Linking CSS to React

1. External CSS Files

Just like in regular HTML, you can link external CSS files to your React components.

```css
/* styles.css */
.my-component {
  background-color: lightblue;
  font-size: 16px;
}
```

```jsx
import React from 'react';
import './styles.css'; // Import the external CSS file

function MyComponent() {
  return (
    <div className="my-component">
      {/* Your component content */}
    </div>
  );
}

export default MyComponent;
```

Linking CSS to React

2. Inline Styles

React allows you to apply styles directly to individual JSX elements using inline styles. This is useful when you want to define styles dynamically or conditionally. Inline styles are defined as JavaScript objects where keys are CSS property names in camelCase:

```jsx
import React from 'react';

function MyComponent() {
  const myComponentStyles = {
    backgroundColor: 'lightblue',
    fontSize: '16px',
  };

  return (
    <div style={myComponentStyles}>
      {/* Your component content */}
    </div>
  );
}

export default MyComponent;
```

Linking CSS to React

3. CSS in JavaScript Libraries

CSS-in-JS libraries, such as styled-components, offer a more powerful and dynamic way to style React components. These libraries allow you to write CSS directly in your JavaScript code.

a. Install styled-components:

```
npm install styled-components
```

Linking CSS to React

3. CSS in JavaScript Libraries

b. Use styled-components to create styled components:

```
import React from 'react';
import styled from 'styled-components';

const StyledComponent = styled.div`
  background-color: lightblue;
  font-size: 16px;
`;

function MyComponent() {
  return (
    <StyledComponent>
      {/* Your component content */}
    </StyledComponent>
  );
}

export default MyComponent;
```

Simple React Component

Welcome to my simple React component!

This is an example of a React component with external CSS.

Styled Components

1. Create a React Project (if not already created):

```
npx create react-app my-styled-components-app
```

1. Install styled-components Package:

```
npm install styled-components
```

1. Import styled-components:

```
import styled from 'styled-components';
```

Styled Components

4. Create Styled Components:

```
const Button = styled.button`
  background-color: #007bff;
  color: white;
  padding: 10px 20px;
  border: none;
  border-radius: 4px;
  cursor: pointer;
`;
```

5. Use Styled Components:

```
function App() {
  return (
    <div>
      <Button>Click Me</Button>
    </div>
  );
}
```

Styled Component Example

- You can create styled components directly in the **same file** where you intend to use them.
- This approach is suitable for small projects or when the styled component is specific to a single component.

```
import React from 'react';
import styled from 'styled-components';

const Button = styled.button`
  background-color: #007bff;
  color: white;
  padding: 10px 20px;
  border: none;
  border-radius: 4px;
  cursor: pointer;
`;

function MyComponent() {
  return (
    <div>
      <Button>Click Me</Button>
    </div>
  );
}
```

Styled Component Example

- For larger projects or when you want to maintain better organization, you can create separate files for your styled components.
- Then, import and use them where needed.

button.jsx

```jsx
import styled from 'styled-components';

export const Button = styled.button`
  background-color: #007bff;
  color: white;
  padding: 10px 20px;
  border: none;
  border-radius: 4px;
  cursor: pointer;
`;
```

MyComponent.jsx

```jsx
import React from 'react';
import { Button } from './Button';

function MyComponent() {
  return (
    <div>
      <Button>Click Me</Button>
    </div>
  );
}
```

Styled Component Example

Dynamic Styling

Using Props for Dynamic Styling:

- Styled-components allows you to apply styles dynamically based on component props. This feature is incredibly useful for creating versatile and responsive designs.

```
const Button = styled.button`
  background-color: ${props => props.primary ? '#007bff' : 'white'};
  color: ${props => props.primary ? 'white' : '#007bff'};
  // Other styles...
`;

// Usage
<Button primary>Primary Button</Button>
<Button>Secondary Button</Button>
```

Dynamic Styling

Example 2 – Conditional Rendering:

```
const Alert = styled.div`
  background-color: ${props => props.success ? 'green' : 'red'};
  color: white;
  // Other styles...
`;

// Usage
<Alert success>Success Alert</Alert>
<Alert>Error Alert</Alert>
```

Dynamic Styling

Responsive Design with Media Queries:

Styled-components seamlessly integrates with media queries for responsive web design. You can adapt your component styles to different screen sizes and orientations.

```
const Heading = styled.h1`
  font-size: 24px;

  @media (max-width: 768px) {
    font-size: 18px;
  }
`;
```

Global Styling and Theming

Theming

- Theming in styled-components allows you to define a set of global styles and variables, ensuring a consistent and cohesive design system across your application.
- It's particularly useful when you want to maintain brand consistency or switch between light and dark themes.

```
import { ThemeProvider } from 'styled-components';
```

Theming

```
const theme = {
  colors: {
    primary: '#007bff',
    secondary: '#ff6600',
  },
  fonts: {
    heading: 'Arial, sans-serif',
    body: 'Roboto, sans-serif',
  },
};

ReactDOM.render(
  <React.StrictMode>
    {/* Wrap your App component with ThemeProvider */}
    <ThemeProvider theme={theme}>
      <App />
    </ThemeProvider>
  </React.StrictMode>,
  document.getElementById('root')
);
```

```
import styled from 'styled-components';

const StyledButton = styled.button`
  background-color: ${props => props.theme.colors.primary};
  color: white;
  padding: 10px 20px;
  border: none;
  border-radius: 4px;
  cursor: pointer;
  font-family: ${props => props.theme.fonts.body};
`;

function MyComponent() {
  return (
    <div>
      <StyledButton>Primary Button</StyledButton>
    </div>
  );
}

export default MyComponent;
```

Global Styling

```
import { createGlobalStyle } from 'styled-components';

const GlobalStyles = createGlobalStyle`
  body {
    margin: 50px;
  }
`;

export default GlobalStyles;
```

Global Styling

index.js

src > index.js > ...

```
21  const root = ReactDOM.createRoot(document.getElementById('root'));
22  root.render(
23    <React.StrictMode>
24      <GlobalStyles/>
25      <ThemeProvider theme={theme}>
26        <App />
27      </ThemeProvider>
28    </React.StrictMode>
29  );
30
```

Nesting and Composition

- Styled-components allows you to nest styled components within each other. This is helpful for creating complex UI structures and maintaining a clean and organized codebase.
- Composition is a technique where you create reusable styled component blocks that can be used in various parts of your application.

Nesting and Composition

```
const Card = styled.div`
  background-color: white;
  border: 1px solid #ccc;
  padding: 16px;
`;

const RoundedCard = styled(Card)`
  border-radius: 8px;
`;
```

Best Practices and Considerations

Structuring Your Styled-Components:

- To maintain code quality and readability, consider organizing your styled-components in a structured manner.
- Group related styles together, either in separate files or within the same file.
- Use meaningful names for your styled components to convey their purpose.

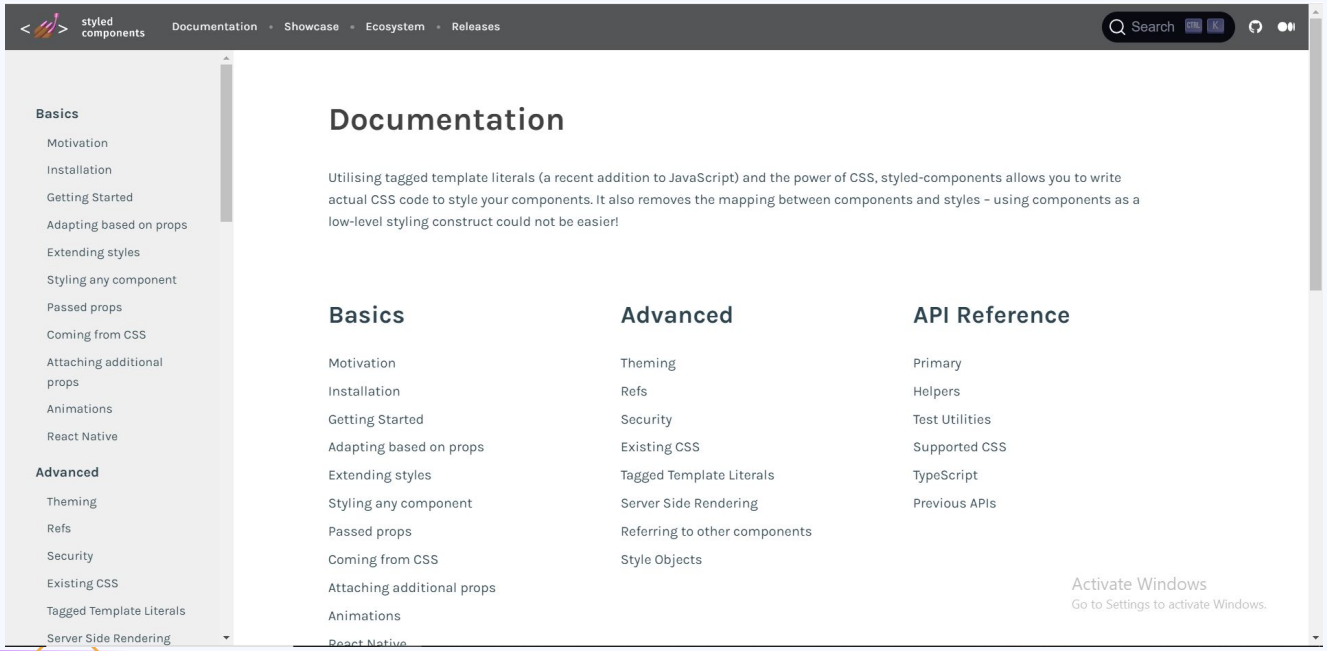
Best Practices and Considerations

Separation of Concerns:

- Keep the concerns of styling and logic separate. Your styled components should focus solely on styling, while logic and functionality belong in React components or hooks.
- Follow the Single Responsibility Principle (SRP) to ensure each component has a clear and distinct purpose.

Styled-Components Documentation

https://styled-components.com/docs



Arrow Function

```jsx
export default function Arrow() {
    const hello = function() {
        return "Hello World!";
    };

    const hello_arrow = () => {
        return "Hello World!";
    };

    // Example of an arrow function with parameters
    const greet = (name) => {
        return `Hello, ${name}!`;
    };

    // Arrow function with a parameter and implicit return
    const implicit_return = (name) => `Hello, ${name}!`;


    return (
        <div>
            <p>{hello()}</p>
            <p>{hello_arrow()}</p>
            <p>{greet('Alice')}</p>
            <p>{implicit_return('Bob')}</p>
        </div>
    );
```

Scope

```
console.log('Component rendering...');

var a = 10;
let b = 20;
const c = 30;

if (true) {
    var a = 40; // Re-declared in the same function scope
    let b = 50; // Block-scoped, different from the outer 'b'
    const c = 60; // Block-scoped, different from the outer 'c'

    console.log(a); // Outputs: 40
    console.log(b); // Outputs: 50
    console.log(c); // Outputs: 60
}

console.log(a); // Outputs: 40 ('var' is function-scoped)
console.log(b); // Outputs: 20 (outer 'b' is unchanged)
console.log(c); // Outputs: 30 (outer 'c' is unchanged)

return (
    <div>
        Open the console to see the output demonstrating variable scope and hoisting.
    </div>
);
```

Event handlers

Event Listening: Event handlers are attached to specific elements to "listen" for particular events on those elements. For example, a button element may have an event handler that listens for click events.

Interactivity: They are fundamental to creating interactive web pages. Without event handlers, a webpage would not be able to respond to user actions like clicks, typing, or mouse movements.

Asynchronous Nature: Event handlers often deal with asynchronous events, meaning they can occur at any time and the code must be prepared to handle these events whenever they happen.

Event handlers

1. **Mouse Events:**
   - **onClick** - Executed when an element is clicked.
   - **onDoubleClick** - Executed when an element is double-clicked.
   - **onMouseDown** - Executed when the mouse button is pressed down over an element.
   - **onMouseMove** - Executed as the mouse is moved over an element.
   - **onMouseOver** - Executed when the mouse enters an element.
   - **onMouseOut** - Executed when the mouse leaves an element.
   - **onMouseUp** - Executed when the mouse button is released over an element.

Event handlers : Mouse Events

```jsx
import React from 'react';

export default function EventHandler() {
  return (
    <div style={{ padding: '20px' }}>

      <button onClick={() => alert('Button Clicked!')}>
        Click me
      </button>

      <button onDoubleClick={() => alert('Button Double Clicked!')}>
        Double Click me
      </button>

      <div
       // onMouseDown={() => console.log('Mouse Down!')}
        //onMouseUp={() => alert('Mouse Up!')}
        style={{ marginTop: '20px', padding: '30px', border: '1px solid black' }}
      >
        Mouse Down & Up here
      </div>

      <div
        //onMouseMove={() => alert('Mouse is moving!')}
        style={{ marginTop: '10px', height: '50px', border: '1px solid black' }}
      >
        Move Mouse Over here
      </div>

      <div
        //onMouseOver={() => alert('Mouse Over!')}
       // onMouseOut={() => alert('Mouse Out!')}
        style={{ marginTop: '10px', height: '50px', border: '1px solid black' }}
      >
        Mouse Over & Out here
      </div>

    </div>
  );
}
```

Event Objects (e)

- The Event Object e represents the event that has occurred.
- Provides access to properties and methods related to the event.
- Can be used to access the target element, prevent default behavior, and more.
- Event object is pivotal in managing user interactions in React applications
    1. e.target - Accessing Event Target
    2. e.target.value - Retrieving Input Values with e.target.value
        - Typically used with form elements like input, textarea, and select.
    3. e.preventDefault() - Preventing Default Behavior
        - Often used with form submissions to prevent page reloads.

Event handlers

2. Keyboard Events:

- **onKeyDown** - Executed when a key is pressed down.
- **onKeyPress** - Executed when a key is pressed.
- **onKeyUp** - Executed when a key is released.

3. Form Events:

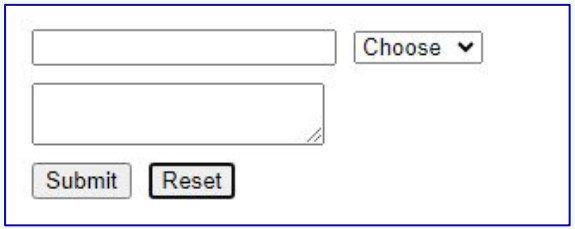
- **onChange** - Executed when the value of an input, select, or textarea is changed.
- **onSubmit** - Executed when a form is submitted.
- **onFocus** - Executed when an element receives focus.
- **onBlur** - Executed when an element loses focus.
- **onInput** - Executed when an input element receives user input.
- **onReset** - Executed when a form is reset.

Event handlers : Keyboard Events

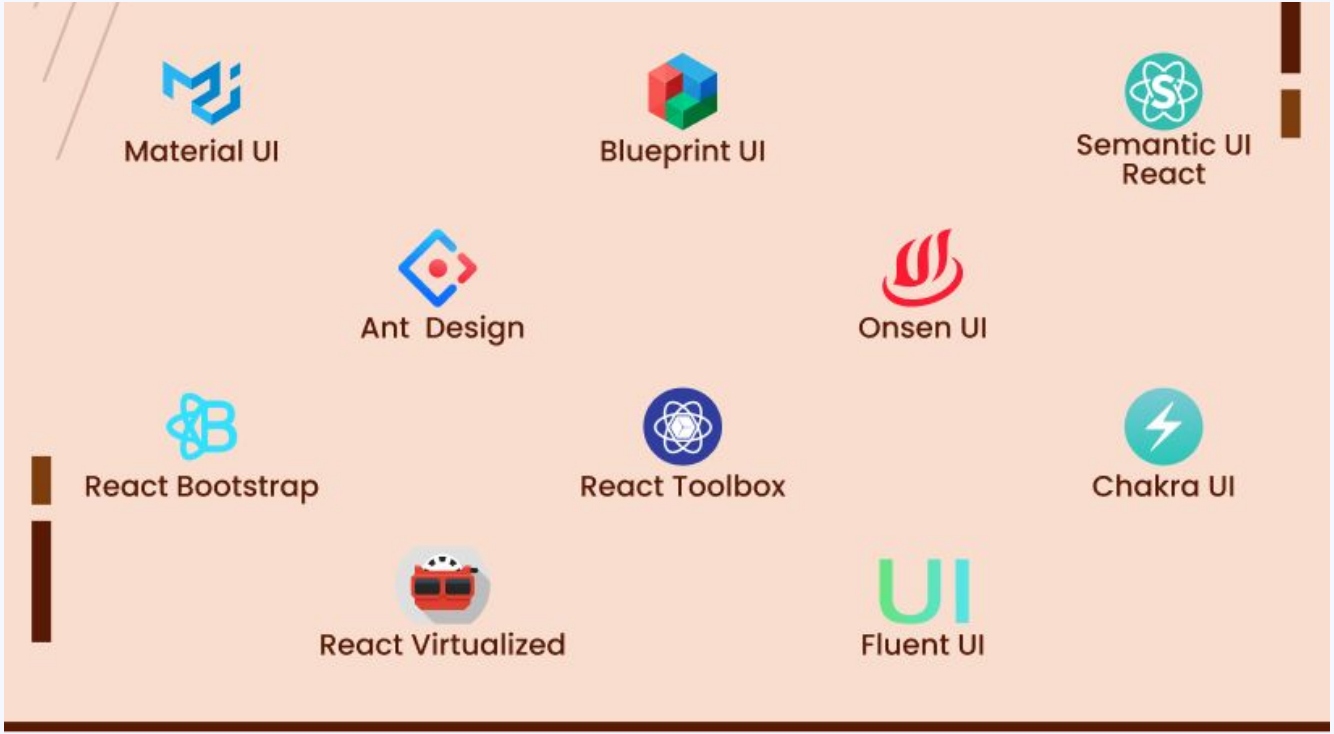
```jsx
import React from 'react';

export default function KeyboardEvents() {
  const handleKeyDown = (e) => {
    console.log('Key Down:', e.key);
  };

  const handleKeyPress = (e) => {
    console.log('Key Press:', e.key);
  };

  const handleKeyUp = (e) => {
    console.log('Key Up:', e.key);
  };

  return (
    <div style={{ padding: '20px' }}>


      <input
        type="text"
        onKeyDown={handleKeyDown} // Executed when a key is pressed down.
        // onKeyPress={handleKeyPress} // Executed when a key is pressed.
        onKeyUp={handleKeyUp} // Executed when a key is released.
        placeholder="Type something..."
        style={{ marginTop: '10px', display: 'block' }}
      />
    </div>
  );
}

```

Event handlers : Form Events

```jsx
import React, { useState } from 'react';

export default function FormEvents() {
  const [inputValue, setInputValue] = useState('');
  const [selectValue, setSelectValue] = useState('');
  const [textareaValue, setTextareaValue] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault(); // Prevents the default form submission.
    alert('Form Submitted');
  };

  const handleInputChange = (e) => {
    console.log('Input Tag:', e.target); // Logs the target input element.
    setInputValue(e.target.value); // Sets the state with the input value.
  };

  const handleReset = () => {
    setInputValue('');
    setSelectValue('');
    setTextareaValue('');
  };

  return (
    <div style={{ padding: '20px' }}>

      <form onSubmit={handleSubmit} onReset={handleReset}>

        <input
          type="text"
          value={inputValue}
          onChange={handleInputChange} // Sets input value and logs the input tag.
          onFocus={(e) => console.log('Input Focused', e.target)}
          onBlur={(e) => console.log('Input Blurred', e.target)}
          onInput={(e) => console.log('User Input Received', e.target.value)}
        />

        <select
          value={selectValue}
          onChange={(e) => setSelectValue(e.target.value)} // Sets select value.
          onFocus={(e) => console.log('Select Focused', e.target)}
          onBlur={(e) => console.log('Select Blurred', e.target)}
          style={{ marginLeft: '10px' }}
        >
          <option value="">Choose</option>
          <option value="option1">Option 1</option>
          <option value="option2">Option 2</option>
        </select>

        <textarea
          value={textareaValue}
          onChange={(e) => setTextareaValue(e.target.value)} // Sets textarea value.
          onFocus={(e) => console.log('Textarea Focused', e.target)}
          onBlur={(e) => console.log('Textarea Blurred', e.target)}
          style={{ display: 'block', marginTop: '10px' }}
        />

        <button type="submit" style={{ marginTop: '10px' }}>Submit</button>
        <button type="reset" style={{ marginLeft: '10px' }}>Reset</button>

      </form>

    </div>
  );
}
```



Introduction to UI Libraries



What are UI Libraries?

- UI Libraries are collections of pre-designed and pre-coded user interface components and styles that can be easily integrated into web applications.
- Benefits:
  - Faster development
  - Consistent design
  - Accessibility
  - Enhanced user experience

Why Use UI Libraries?

- **Efficiency:** Save time and effort by using pre-built components.
- **Consistency:** Ensure a cohesive design throughout your application.
- **Accessibility:** Many libraries prioritize accessibility features.
- **Community:** Benefit from a community of developers and contributors.
- **Flexibility:** Customize and extend components as needed.

Introduction to Material-UI

- Material-UI is a popular UI library for React that follows the principles of Google's Material Design.
- Features:
  - Comprehensive set of components
  - Easy theming and customization
  - Accessible design
  - Active community and support
- Website: https://mui.com/

Key Features of Material-UI

- **Material Design:** Based on Google's Material Design guidelines for a modern and sleek UI.
- **Component Library:** A wide range of components like buttons, cards, and navigation bars.
- **Responsive:** Components are designed to work seamlessly on various screen sizes.
- **Theming:** Easily create and apply custom themes for your app.
- **Accessibility:** Built-in accessibility features for inclusive design.
- **Active Community:** A large and active community of users and contributors.

Getting Started with Material-UI

1. Installation

```
npm install @mui/material @mui/icons-material
```

1. Installing necessary Dependencies

```
npm install @emotion/react @emotion/styled
```

Getting Started with Material-UI

3. Creating a Material-UI Component

```
import React from 'react';
import Button from '@mui/material/Button';

function MyButton() {
  return (
    <Button variant="contained" color="primary">
      Click Me
    </Button>
  );
}

export default MyButton;
```

Introduction to Material-UI Icons

Material-UI Icons is a library that provides a wide range of pre-designed icons for use in React applications.

https://mui.com/material-ui/material-icons/

1. **Installing Material-UI Icons**

```
npm install @mui/icons-material
```

1. **Importing Material-UI Icons**

```
import HomeIcon from '@mui/icons-material/Home';
```

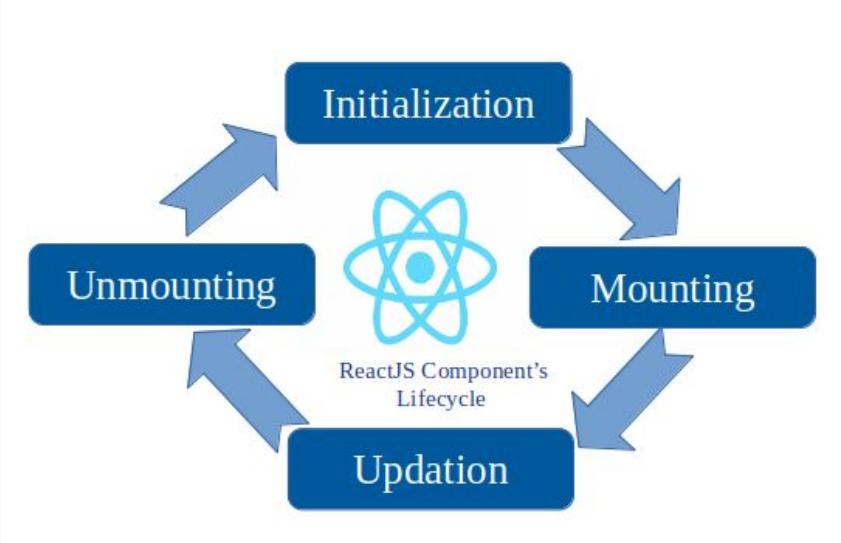
Adding an Image

import Logo from '../Styles/images/download.jpg';

```
<img src={Logo} width="500px" height="500px" alt="logo" />
```

Homework : Create an attractive landing page with a styled navigation bar

Understanding the React Component Lifecycle



Class Components - Recap

- Class components based on JavaScript ES2015. It extends Component base class from React and it gives way to the react life cycle methods and add a render function which is use for return the react elements.

```
class App extends Component {
  render() {
    return (
      <View>
        <Text>Hello, world!</Text>
      </View >
    )
  }
};
```

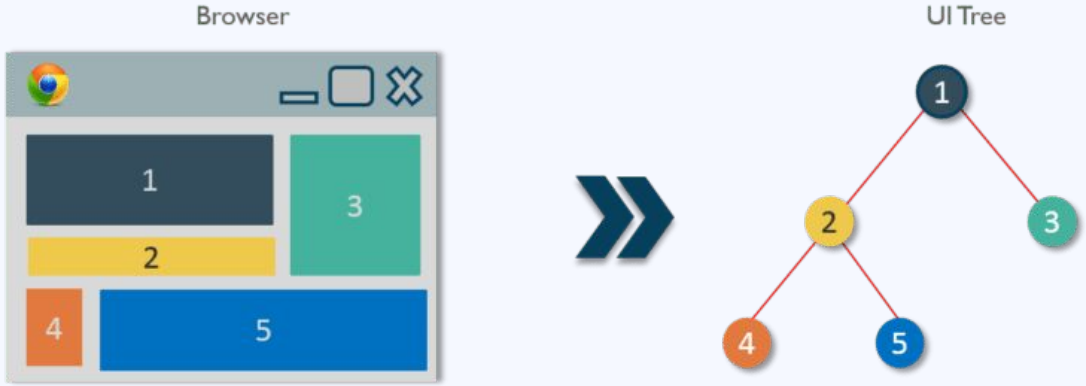
Functional Components - Recap

- Functional components based on simple or plain JavaScript. Functional components can't maintain their own state that's why sometimes we can say it stateless components. It is just accept the props as an argument and return the react elements.

```
function Hello(props) {
  return <h2>Hello Dear, {props.name}</h2>;
}
```

React Components - Recap

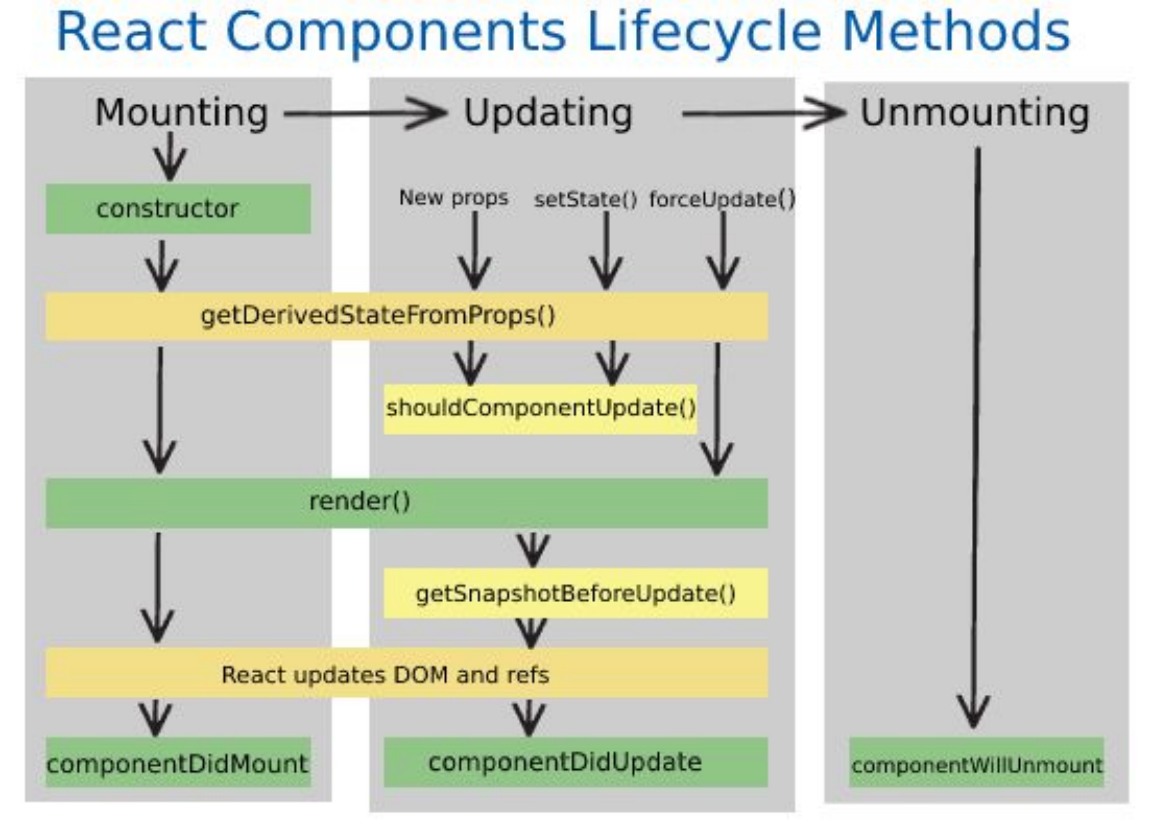
- React components are reusable, self-contained units of a user interface.
- Components can be thought of as custom HTML elements.
- They encapsulate both the UI and the logic related to it.
- Components make it easier to manage and maintain complex user interfaces.



React Component Lifecycle

- The React component lifecycle is a series of phases that a component goes through during its existence.
- These phases control when and how a component is created, updated, and destroyed.
- Understanding the component lifecycle helps us manage state, perform side effects, and optimize rendering.
- **Three Main Phases :**
    1. Mounting Phase
    2. Updating Phase
    3. Unmounting Phase

React Component Lifecycle



Mounting Phase

- The Mounting phase is the initial phase when a component is created and inserted into the DOM.
- Key lifecycle methods in this phase:
  - **constructor:** Called when the component is initialized.
  - **getDerivedStateFromProps:** Used for updating the component's state based on props.
  - **render:** Defines the component's UI.
  - **componentDidMount:** Called after the component is added to the DOM.

Mounting Phase

The render() method is required and will always be called, the others are optional and will be called if you define them.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    // Initialize state and bindings here
  }

  static getDerivedStateFromProps(nextProps, prevState) {
    // Update state based on props
    if (nextProps.someValue !== prevState.someValue) {
      return { someValue: nextProps.someValue };
    }
    return null; // No state update necessary
  }

  render() {
    return (
      // JSX code for the component's UI
    );
  }

  componentDidMount() {
    // Perform actions after the component is mounted
  }
}
```

Mounting Phase : Constructor()

- The constructor method is called first when a component is created.
- It's used for initializing state, setting up initial values, and binding event handlers.

```
constructor() {
  super();
  this.state = {
    count: 0,
  };
  this.handleClick = this.handleClick.bind(this);
}
```

Mounting Phase : getDerivedStateFromProps()

- In React, when a component is being created and inserted into the DOM during the Mounting phase, you can use a lifecycle method called getDerivedStateFromProps().
- getDerivedStateFromProps() is a static method that allows you to update the component's state based on changes in its props before the initial render.
- It's commonly used for initializing state values derived from props.
- Unlike other lifecycle methods, getDerivedStateFromProps() does not have access to the component's instance; it's a pure function that receives props and state as arguments and returns an object to update the state.

Mounting Phase : getDerivedStateFromProps()

```
static getDerivedStateFromProps(nextProps, prevState) {
  // Compare the current props with nextProps
  if (nextProps.initialValue !== prevState.initialValue) {
    // If they are different, update the state
    return {
      derivedValue: nextProps.initialValue,
    };
  }
  // If no update is needed, return null
  return null;
}
```

Mounting Phase : Render()

- The render method defines the component's UI by returning JSX.
- It should be a pure function, meaning it doesn't have side effects.

```
render() {
  return (
    <div>
      <p>Count: {this.state.count}</p>
      <button onClick={this.handleClick}>Increment</button>
    </div>
  );
}
```

Mounting Phase : componentDidMount()

- The componentDidMount method is called after the component is added to the DOM.
- It's commonly used for data fetching, setting up subscriptions, or interacting with the DOM.

```
componentDidMount() {
  // Fetch data from an API
  fetch('https://api.example.com/data')
    .then(response => response.json())
    .then(data => {
      this.setState({ data });
    });
}
```

Replicates in Functional Component

- You can replicate the **componentDidMount** lifecycle method in functional components using the **useEffect** hook. The code within **useEffect** will execute after the component is mounted:

```
import React, { useEffect } from 'react';

function MyFunctionalComponent() {
  useEffect(() => {
    // Code to run after component is mounted
    return () => {
      // Cleanup code if needed (equivalent to componentWillUnmount)
    };
  }, []); // Empty dependency array means it only runs once after mounting

  return (
    // JSX for the component's UI
  );
}
```

Updating Phase

- The Update phase in React occurs when a component's state or props change. During this phase, React evaluates the need to update and makes necessary updates to the component.
- Key lifecycle methods in the Update phase:
  - **getDerivedStateFromProps():** Used for updating the component's state based on changes in props.
  - **shouldComponentUpdate():** Decides whether the component should re-render.
  - **render():** Defines the updated UI.
  - **getSnapshotBeforeUpdate():** Captures a snapshot of the current UI before changes are made to the DOM.
  - **componentDidUpdate():** Called after the component updates in the DOM.

Updating Phase

The render() method is required and will always be called, the others are optional and will be called if you define them.

```
class UpdateExample extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0,
    };
  }

  static getDerivedStateFromProps(nextProps, prevState) {
    // Update state based on props changes
    if (nextProps.initialCount !== prevState.count) {
      return { count: nextProps.initialCount };
    }
    return null;
  }

  shouldComponentUpdate(nextProps, nextState) {
    // Decide whether to re-render based on conditions
    return this.state.count !== nextState.count;
  }

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
      </div>
    );
  }

  getSnapshotBeforeUpdate(prevProps, prevState) {
    // Capture a snapshot before the update
    return prevState.count;
  }

  componentDidUpdate(prevProps, prevState, snapshot) {
    // Perform actions after the component updates
    console.log(`Component updated from ${snapshot} to ${this.state.count}`
  }
}
```

Updating Phase : getDerivedStateFromProps()

- **getDerivedStateFromProps()** is a static method used in the Update phase.
- It allows you to update the component's state based on changes in props before rendering.

```
class MyComponent extends React.Component {
  static getDerivedStateFromProps(nextProps, prevState) {
    // Update state based on props changes
    if (nextProps.someValue !== prevState.someValue) {
      return { someValue: nextProps.someValue };
    }
    return null;
  }

  render() {
    return (
      <div>
        {/* JSX for the component's UI */}
      </div>
    );
  }
}
```

Updating Phase : shouldComponentUpdate()

- **shouldComponentUpdate()** is a method used in the Update phase.
- It determines whether the component should re-render based on certain conditions.

```
class MyComponent extends React.Component {
  shouldComponentUpdate(nextProps, nextState) {
    // Decide whether to re-render based on conditions
    return this.props.someValue !== nextProps.someValue;
  }

  render() {
    return (
      <div>
        {/* JSX for the component's UI */}
      </div>
    );
  }
}
```

Updating Phase : render()

- **render()** is a fundamental method used in the Update phase.
- It defines the updated UI of the component.

```
class MyComponent extends React.Component {
  render() {
    return (
      <div>
        {/* Updated JSX for the component's UI */}
      </div>
    );
  }
}
```

Updating Phase : getSnapshotBeforeUpdate()

- **getSnapshotBeforeUpdate()** is a method used in the Update phase.
- It captures a snapshot of the current UI before changes are made to the DOM.

```
class MyComponent extends React.Component {
  getSnapshotBeforeUpdate(prevProps, prevState) {
    // Capture a snapshot before the update
    return prevState.someValue;
  }

  render() {
    return (
      <div>
        {/* JSX for the component's updated UI */}
      </div>
    );
  }
}
```

Updating Phase : componentDidUpdate()

- **componentDidUpdate()** is called after the component updates in the DOM.
- It's used for performing actions after the update, such as interacting with the DOM or managing side effects.

```
class MyComponent extends React.Component {
  componentDidUpdate(prevProps, prevState, snapshot) {
    // Perform actions after the component updates
    console.log(`Component updated from ${snapshot} to ${this.state.someVal
  }

  render() {
    return (
      <div>
        {/* JSX for the component's updated UI */}
      </div>
    );
  }
}
```

Replicates in Functional Component

- Replicate **componentDidUpdate** using the **useEffect** hook with a dependency array that includes the variables you want to monitor for changes.

```
import React, { useEffect, useState } from 'react';

function MyFunctionalComponent() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    // Code to run whenever 'count' changes
    console.log('Count updated:', count);
  }, [count]); // Runs whenever 'count' changes

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

Unmounting Phase

- The Unmounting phase occurs when a component is removed from the DOM.
- The key lifecycle method in this phase is **componentWillUnmount.**
- componentWillUnmount is called just before the component is removed from the DOM.
- It's used for cleanup tasks.

```
componentWillUnmount() {
  // Clean up resources, like canceling network requests or removing event
  this.cancelRequest();
}
```

Replicates in Functional Component

- Replicate the cleanup behavior of **componentWillUnmount** by returning a cleanup function from **useEffect**. This function will be called when the component is unmounted:

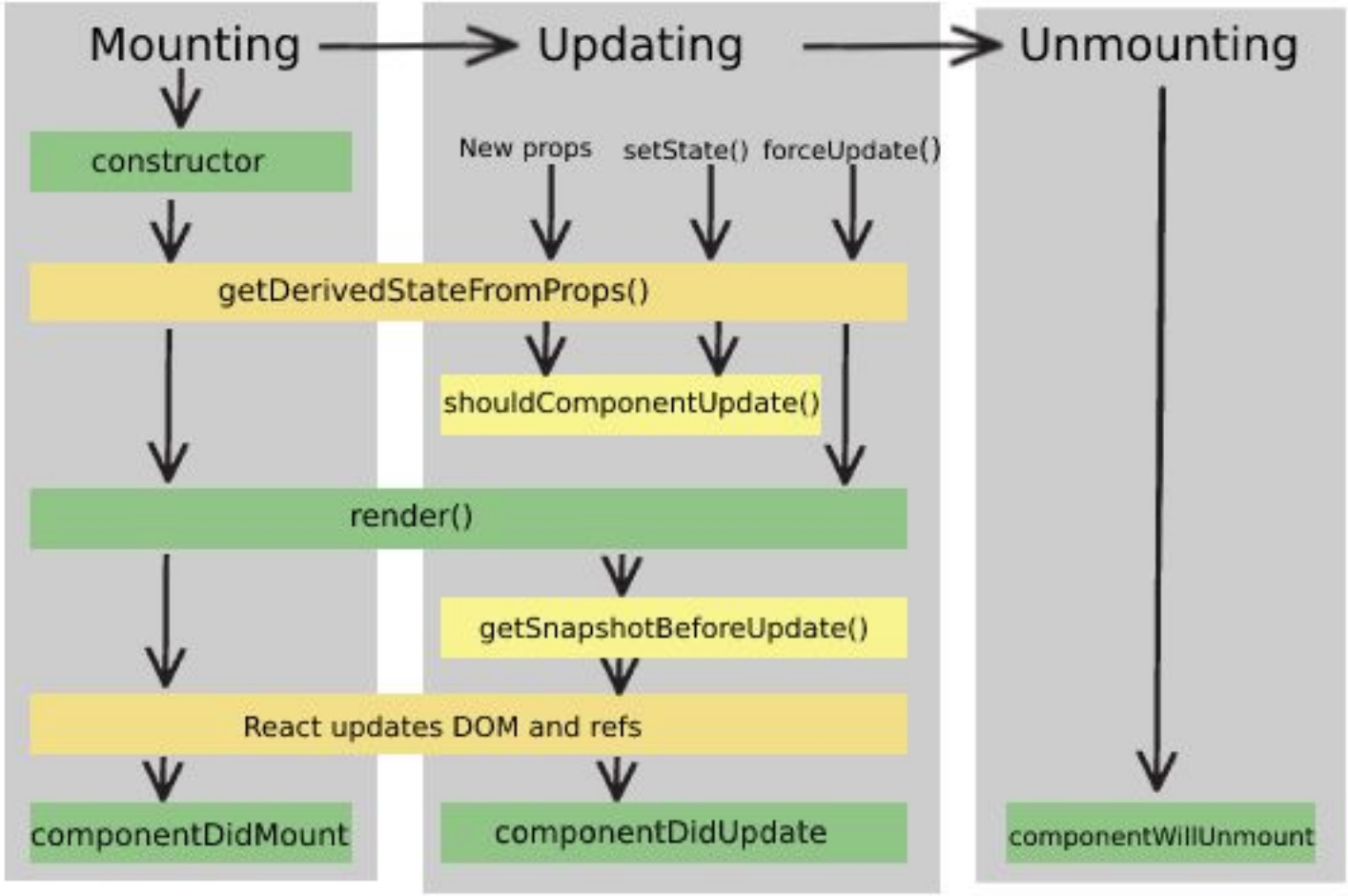
```
import React, { useEffect } from 'react';

function MyFunctionalComponent() {
  useEffect(() => {
    // Code to run after component is mounted

    return () => {
      // Cleanup code when component is unmounted
    };
  }, []);

  return (
    // JSX for the component's UI
  );
}
```

Error Handling

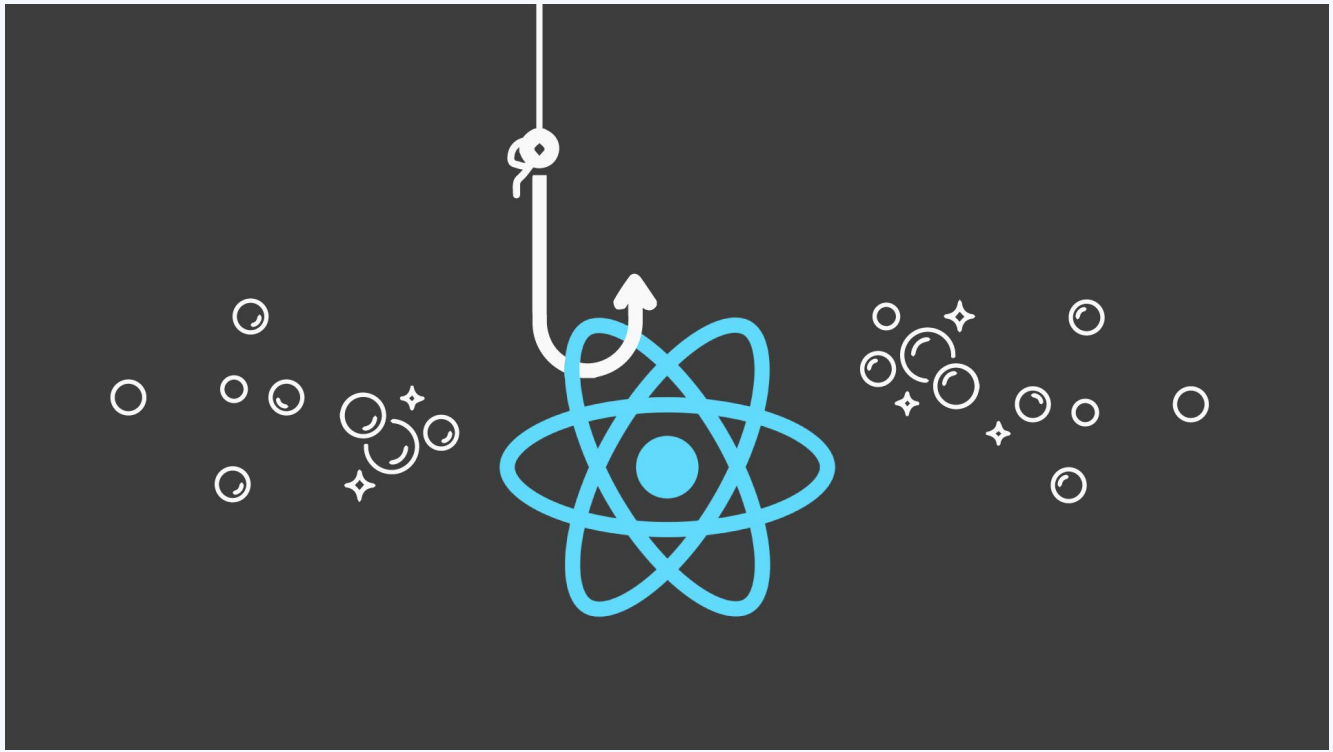
- React provides error handling methods like **componentDidCatch.**
- They are used to handle errors that occur within a component.

```
class ErrorBoundary extends React.Component {
  constructor() {
    super();
    this.state = { hasError: false };
  }

  componentDidCatch(error, info) {
    // Handle errors here
    this.setState({ hasError: true });
  }

  render() {
    if (this.state.hasError) {
      return <div>Error occurred!</div>;
    }
    return this.props.children;
  }
}
```

React Components Lifecycle Methods



React Hooks



Hooks

A hook in React is a special kind of function that allows you to "hook into" React features from within function components

- useState
- useEffect
- useContext
- useReducer
- useCallback
- useMemo
- useRef
- useLayoutEffect

State

State refers to the local state of a component. It is data that dictates how a component renders and behaves. Unlike props, the state is changeable, but there's a proper way to change it.

Initialization: State is often initialized using the useState hook in functional components.

Changing State: In functional components, you use the setState function returned by the useState hook.

State

Rerender: Whenever the state changes, the component re-renders. It's one of the reasons why React components can be dynamic.

Local and Encapsulated: Each component has its own state. It's not accessible to any component other than the one that owns and sets it.

useState

- Allows functional components to maintain local state
- useState hook, returns an array with two elements:
  - The current state value
  - A function to update that state value.
    - Eg:

```
const [count, setCount] = useState(0);
```

- count is the first item, which is the current state value. It is initialized to 0.
- setCount is the second item, which is a function that you can use to update the value of count.

State - Example

```jsx
import React, { useState } from 'react'

export default function UseState() {

  const [count, setCount] = useState(0);

  return (
    <div>
        <h1>UseState Example - Counter</h1>
        <p>You clicked {count} times</p>
        <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  )
}
```

useState Example

2. Strings with useState

```
export default function StringsWithUseState() {
    const [text, setText] = useState("Hello, World!");

    return (
        <div>
            <p>{text}</p>
            <button onClick={() => setText(text.split('').reverse().join(''))}>
                Reverse Text
            </button>
            <button onClick={() => setText(text.toUpperCase())}>
                Uppercase
            </button>
            <button onClick={() => setText(text.toLowerCase())}>
                Lowercase
            </button>
        </div>
    );
}
```

useState Example

3. **Arrays with useState**

```jsx
const [items, setItems] = useState(['Apple', 'Banana', 'Cherry']);

const addItem = () => setItems([...items, 'Date']);

const removeItem = (index) => {
  setItems(items.filter((_, i) => i !== index));
};

const updateItem = (index) => {
  // Assuming you want to update item in some manner
  const newItems = [...items];
  newItems[index] = 'Grape';
  setItems(newItems);
};

return (
  <div>
    <h1>Arrays With UseState Examples</h1>
    <ul>
      {items.map((item, index) => (
        <li key={index}>
          {item}
          <button onClick={() => removeItem(index)}>Remove</button>
          <button onClick={() => updateItem(index)}>Update</button>
        </li>
      ))}
    </ul>
    <button onClick={addItem}>Add Item</button>
  </div>
);
```

useState Example

4. Objects with useState

```jsx
import React, { useState } from 'react';

export default function ObjectsWithUseState() {
  // Initialize state
  const [user, setUser] = useState({ name: 'John', age: 30 });

  // Function to handle name change
  const changeName = () => {
    setUser(prevUser => ({ ...prevUser, name: 'Jane' }));
  };

  // Function to handle age increment
  const incrementAge = () => {
    setUser(prevUser => ({ ...prevUser, age: prevUser.age + 1 }));
  };

  return (
    <div>
      <h1>Objects With UseState Examples</h1>
      <p>{user.name} is {user.age} years old</p>
      <button onClick={changeName}>Change Name</button>
      <button onClick={incrementAge}>Increment Age</button>
    </div>
  );
}
```

Thanks!